NAVAL
POSTGRADUATE
SCHOOL

MONTEREY, CALIFORNIA

THESIS

DEVELOPING A CONCEPTUAL ARCHITECTURE FOR A GENERALIZED AGENT-BASED MODELING ENVIRONMENT (GAME)

by

Christian T. Nguyen

March 2008

Thesis Advisor:	Dan Dolk
Second Reader:	Albert Baretto

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** March 2008	**3. REPORT TYPE AND DATES COVERED** Master's Thesis

4. TITLE AND SUBTITLE Developing a Conceptual Architecture for a Generalized Agent-based Modeling Environment (GAME)	**5. FUNDING NUMBERS**
6. AUTHOR(S) Christian T. Nguyen	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited	**12b. DISTRIBUTION CODE** A

13. ABSTRACT (maximum 200 words)

Agent-based technology is a relatively new, but rapidly proliferating decision technology. The relative immaturity of ABM software often requires significant programmer expertise in model representation and implementation. This limits potential users and their ability to utilize the software. We develop a high level conceptual architecture for an agent-based modeling environment which overcomes this limitation. This thesis defines a taxonomy of agents based on commonly accepted agent characteristics, reviews six of the most popular software platforms for agent-based model development, and maps their relationship to the taxonomy. Past modeling advances in the operations research and management science (OR/MS) domains indicate that a more generalized environment is possible. A conceptual architecture for a generalized agent-based modeling environment (GAME) based upon design principles from OR/MS systems was created that would overcome some, if not all, of these obstacles. The GAME architecture incorporates higher-level model representations separate from solver code, a library of transformation procedures, reusable model libraries and a robust language or equivalent interface for specifying experimental design procedures. Rapid technology development would allow for agent-based modeling software that subsequently benefits a much wider range of stakeholders than is currently the case. Finally, embedding GAME in an even higher-level integrated decision technology environment (IDTE) would facilitate the integration of computational and analytical modeling.

14. SUBJECT TERMS Agent-based modeling and simulation, decision technology, modeling environment, model representation, solver, model reusability.	**15. NUMBER OF PAGES** 77
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

DEVELOPING A CONCEPTUAL ARCHITECTURE FOR A GENERALIZED AGENT-BASED MODELING ENVIRONMENT (GAME)

Christian T. Nguyen
Lieutenant, United States Navy
B.S., LSU, 1995

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN INFORMATION TECHNOLOGY MANAGEMENT

from the

NAVAL POSTGRADUATE SCHOOL
March 2008

Author:	Christian T. Nguyen
Approved by:	Dan Dolk Thesis Advisor
	Albert “Buddy” Barreto Second Reader
	Dan Boger Chairman, Department of Information Sciences

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Agent-based technology is a relatively new, but rapidly proliferating decision technology. The relative immaturity of ABM software often requires significant programmer expertise in model representation and implementation. This limits potential users and their ability to utilize the software. We develop a high level conceptual architecture for an agent-based modeling environment which overcomes this limitation. This thesis defines a taxonomy of agents based on commonly accepted agent characteristics, reviews six of the most popular software platforms for agent-based model development, and maps their relationship to the taxonomy. Past modeling advances in the operations research and management science (OR/MS) domains indicate that a more generalized environment is possible.

A conceptual architecture for a generalized agent-based modeling environment (GAME) based upon design principles from OR/MS systems was created that would overcome some, if not all, of these obstacles. The GAME architecture incorporates higher-level model representations separate from solver code, a library of transformation procedures, reusable model libraries and a robust language or equivalent interface for specifying experimental design procedures. Rapid technology development would allow for agent-based modeling software that subsequently benefits a much wider range of stakeholders than is currently the case. Finally, embedding GAME in an even higher-level integrated decision technology environment (IDTE) would facilitate the integration of computational and analytical modeling.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to thank my wife Mai and son Dominic for their support and love because they have made this experience worthwhile. I would also like to thank Professor Dan Dolk for his direction and guidance. Without him, this process could not have been possible. Finally to my friends, your friendship here at NPS I will cherish for the rest of my life, fair winds and following seas.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. PROBLEM CHARACTERISTIC

We are now in an age where information is growing at an exponential rate and is more complex to interpret than ever before. Decision Support Systems (DSS) is one of the primary technologies for keeping up with this growth. There are numerous computer-based decision technologies available including data-driven, document-driven, model-driven and knowledge-driven decision support software. These software systems are used to help decision-makers in their decision-making processes.

Data-driven DSS supply decision-markers with data transformed into useful information through the medium of data warehouses, on-line analytical processing (OLAP), data mining and data visualization. Document-driven DSS rely upon search technology for retrieving and coordinating unstructured data in the form of documents. Model-driven DSS provide analytical capabilities in the form of operations research and management science techniques such as optimization and simulation that help us better see and comprehend the overall decision landscape. Knowledge-driven DSS help us facilitate knowledge flow using technologies such as expert systems and computational organizations.

Most off the shelf DSS software systems suffer from one major limitation, however, namely they are geared towards one specific technology, for example, discrete event simulation for logistics or neural networks for pattern identification. As useful as these may be for their intended application domain, it is possible to foresee that being able to combine these technologies in a single application can extend their utility even further. When used intelligently and in conjunction with one another, these DSS technologies can lead to more effective, and sometimes more efficient, decision making. Thus, our grand objective is to investigate how to implement an Integrated Decision Technologies Environment (IDTE), which provides access to various DSS platforms and makes it relatively easy to understand and use them.

The subject of this research is to examine one specific type of decision technology, agent-based modeling and simulation (ABMS), which has become more and more prevalent over the past decade. Specifically, we will design a conceptual architecture for coordinating existing ABMS software, which will hopefully have the effect of making this technology more accessible to non-programming personnel as well as facilitating integration with other decision technologies to expand the scope of ABMS applicability. This higher-level approach we call a Generalized Agent-based Modeling Environment (GAME), which we see as one important component in an overall IDTE.

B. METHODOLOGY

Agent-based modeling and simulation (ABMS) is particularly heavily used for social science objectives. Social sciences seek not only to understand how individuals behave but how their interaction with others results in large-scale outcomes. It looks at systems that consist of interacting agents who exhibit emergent properties that result from interactions with other agents in ways that cannot be predicted simply by looking at its properties. If the interaction of the agents is contingent upon past experience, then mathematical analysis is limited in its ability to derive dynamic consequences. An alternative to conventional analytical modeling in this case is ABMS.

Software systems for building and analyzing ABMS have proliferated significantly over the past decade as ABMS has become a more popular analytical technique. As such, we are interested in how to incorporate this technology into an IDTE. The first step in this process is to develop a taxonomy of ABMS software systems, which we develop from a software engineering perspective. We then survey existing software platforms for ABMS, identify six of the most popular systems, and show how they fit into the taxonomy. From this vantage point, we then highlight the limitations of existing ABMS software, and develop an overarching conceptual architecture which provides a much more generalized interface for ABMS applications than existing platforms.

C. ORGANIZATION OF THESIS

This chapter provides background information regarding agent-based technologies and the desirability of integrating them into an IDTE. Chapter II presents a taxonomy of six types of agents, based upon a software engineering perspective of their underlying software environments. Chapter III surveys six of the most popular ABMS software platforms: NetLogo, Swarm, AnyLogic, Repast, MASON, and Ascape, to see what features they provide and the relationship they have with the agent taxonomy of Chapter II. Chapter IV creates a conceptual framework for a

Generalized Agent-based Modeling Environment (GAME), which overcomes the programmer-oriented nature of the system surveyed in Chapter III. We will also look at the relationship between the selected software and the conceptual framework. Finally, Chapter V summarizes the importance of agent-based technologies, why they should be integrated into an IDTE, and necessary components for realizing this integration.

II. TAXONOMY OF AGENTS

A. AGENT-BASED MODELING

1. Introduction

Different developers have been using the slogan of Agent-Based Modeling in very different disciplines such as complexity science, artificial intelligence, game theory, etc. Currently there are no universally accepted definitions in this area. People still discuss what properties objects need to have to be considered an agent, for example, proactive and/or reactive behaviors, spatial awareness, the ability to learn and adapt, social skills, some form of intelligence, etc.

Everyone does agree that agents have dynamic behaviors. This is why a modeler defines behavior at the individual level, with the understanding that global behavior forms from the interactions amongst the many individuals, each having their own rules, living together in an environment and interacting with it and each other. This is why ABM is also known as bottom-up modeling. The Agent-based approach offers at least two major advantages:

(1) it is more general and powerful because it allows us to capture more complex structures and dynamics;

(2) it is easier to construct a model when there is little knowledge about global interdependencies.

Agents and multi-agent system offer adaptability, scalability, distribution, fault tolerance, intelligence and autonomy. The notion of agent has been discussed in many different contexts, and many classifications have been proposed. These are usually based on philosophical aspects of agents rather than on any specific control flow followed in the agent model. To be successful, this technology must be philosophically appealing, computationally affordable, easy to develop and effective at solving some classes of problems. Agents should exhibit design in the form of develop, implement, run and debug properties. A taxonomy of software agents based on agent-oriented programming language and platform will be proposed. The taxonomy takes into account internal issues of agents, such as deliberative control flow in addition to philosophical aspects such as intelligence, reactivity and sociability.

B. TAXONOMY

1. Types of Agents

Six types of agents were identified after analyzing several languages and platforms for developing and deploying of agents. These agents are categorized based on the characteristics that they emphasized:

- Commitment agents
- Event-driven agents
- Goal-directed agents
- Software-integration Agents
- Hierarchy-based agents
- Task and Communication-agents

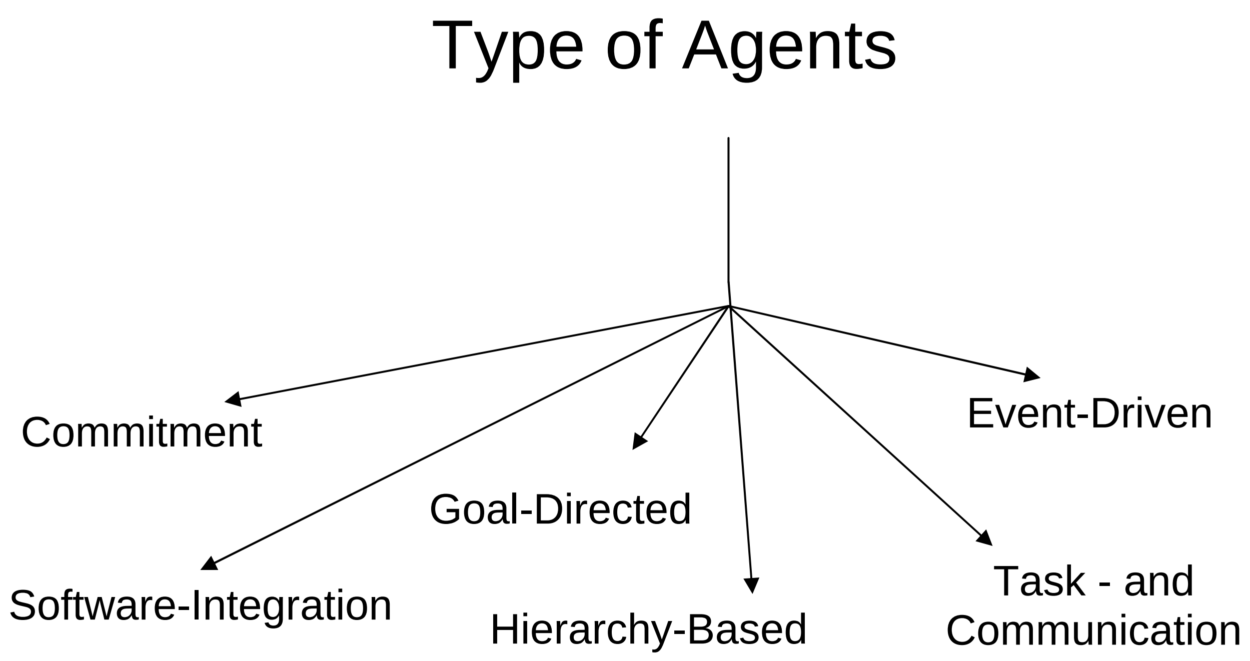


Figure 1. Taxonomy of Agent-Oriented Programming Languages (After [1]).

C. DESCRIPTION OF AGENT TYPES

1. Commitment Agents

A Commitment Agent is “committed” to performing acts in the future according to its beliefs, capabilities and the commitments already taken. Programmers can articulate beliefs, capabilities and rules to formulate new commitments. Time is an essential element for these agents. A time stamp is placed on each belief to determine when the belief will become true and this belief remains in effect until a more recent belief contradicts the old one. In addition, commitments have time stamps to ensure that they will be performed accurately. A Commitment Agent iterates between receiving messages, updating beliefs and commitments, and performing commitments in a timely matter.

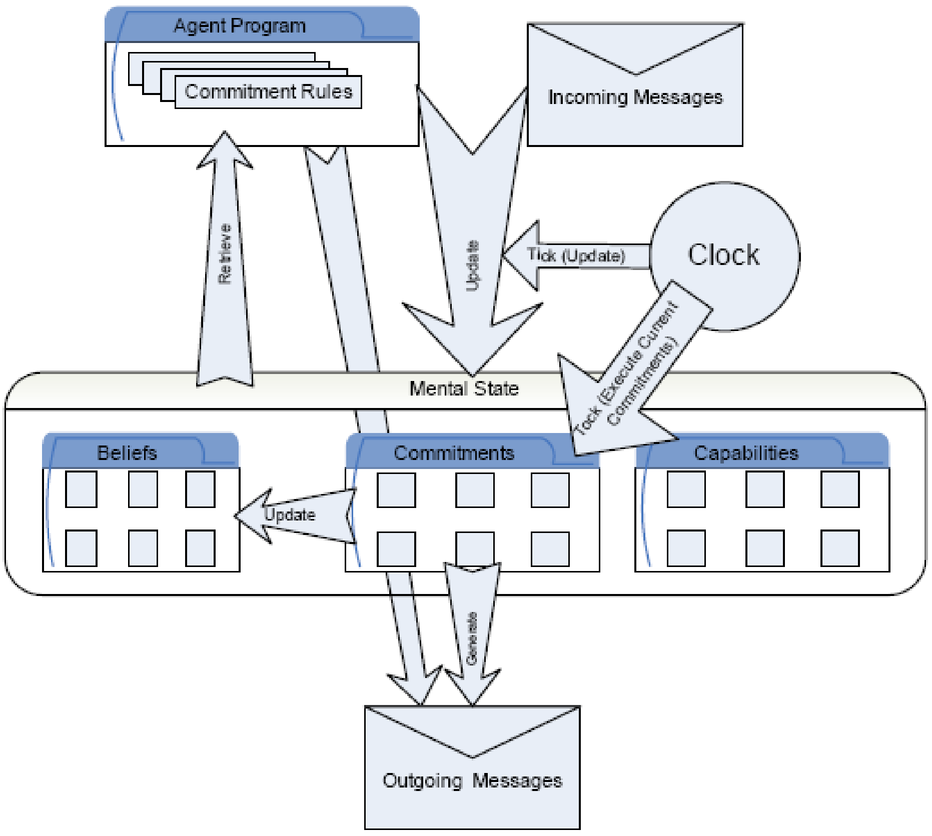


Figure 2. Components of Commitment Agents and their Interactions (From [1]).

2. Event-driven Agents

This type of agent is most popular in the Belief, Desire, Intention (BDI) paradigm of agents. Beliefs are used to represent their knowledge about themselves and the world. As the name suggests, events are very important to these agents. An Event represents the occurrence of a situation that the agent must deal with. There are various events that will change the belief of the agent, its perception or reception of a message. In this case the agent must have the property of being reactive since it can react depending upon its circumstances. Proactive behavior is accomplished by means of goals and they are internally represented by events. Goals are achieve a state, perform an act, maintain state, or query information. Plans are used to handle events, and are procedures that represent the behavioral component of the agent. Plans state what events an agent can handle and under what applicable conditions.

The basic operation of an event-driven agent is to update its beliefs and event queue and select an event to handle. It will then look for plans that will handle the event, and subsequently form the relevant plans. One of these plans will be selected, and either pushed to the top of the current intention pile or put into a newly created pile. Finally, one intention will be executed, thus updating beliefs, sending messages and/or generating new events.

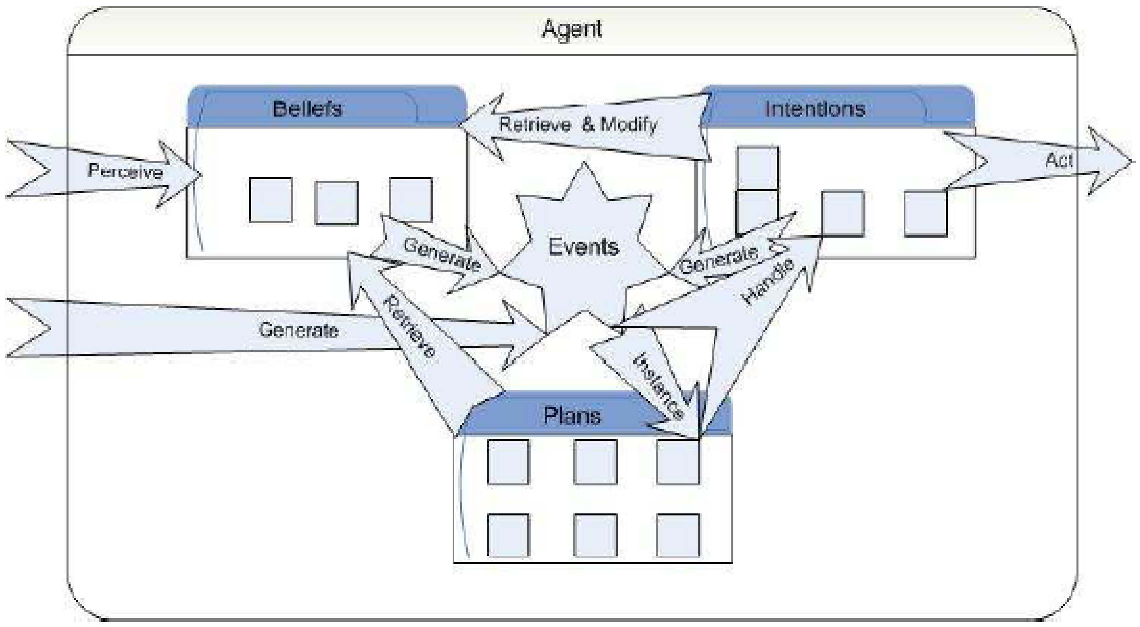


Figure 3. Event-Driven Agent (From [1]).

3. Goal-directed Agents

Goal-directed agents are similar to event-driven agents in that they are constructed on beliefs, goals or plans (BDI-style agent) but they do not work internally the same. Event-driven agents handle the relevant events that occur whereas goal-directed agents satisfy their goals via practical reasoning rules. There are two kinds of practical reasoning rules: goal planning and plan revision. Goal planning rules are used to achieve goals whereas plan revision rules are used to revise plans from a plan base. Both have preconditions which indicate when they can be functional.

A goal-directed agent tries to find goal planning rules that will satisfy it goals, then looks for revision rules that will refine its plans, and finally, chooses and executes a plan. A goal-directed agent can be thought of as an enhanced Prolog program that combines logic programming and procedural knowledge. Because beliefs can be implemented like a Prolog program (both facts and rules), the agent can infer new facts from previous ones. This gives agents a primitive reasoning and learning capability.

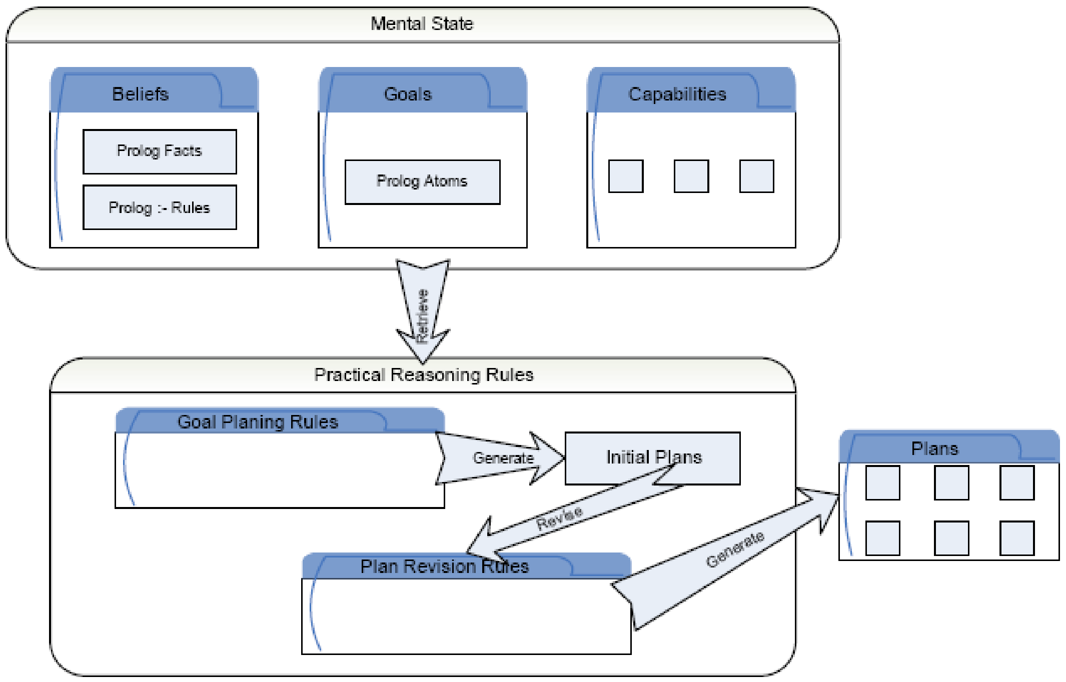


Figure 4. Goal-Oriented Agent (From [1]).

4. Software Integration Agents

The main purpose of this approach is the agentization of legacy code that is adapting legacy code from other applications to serve as the basis for agent behavior. In order to evolve legacy code for use by an agent, the legacy code's main features must be described, for example, in terms of the data types and functions that the software manages. There are three steps in this process:

(1) Retrieve messages and updates its state.

(2) Decides the status of each action in the set of actions in accordance with its program.

(3) Decide the actions that can be executed concurrently.

Legacy code must be described in terms of its data types, its functions and its component operators in order to build new data types.

Agents govern legacy code by utilizing agent programs, which allow them to access their legacy code, getting useful needed data that represents the state of the agent. Agents combine arbitrary code by determining the actions that it can perform by means of its program, and executing these actions by calling the suitable functions of the legacy code. The state of the action allows access to heterogeneous data structures and legacy software code. A body of code in any suitable programming language implements each action, while also defining its precondition and effects. Thus, the agent is somewhat in charge of its actions.

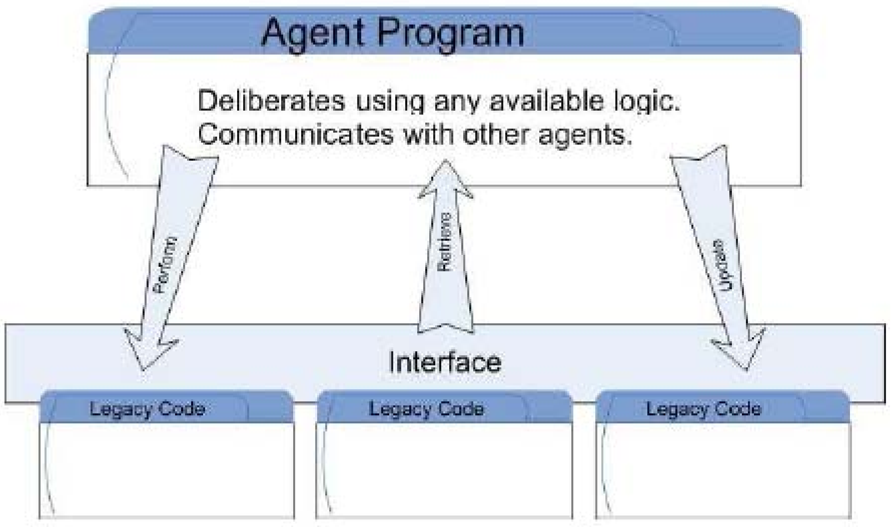


Figure 5. Software-Integration Agent (From [1]).

5. Hierarchy-based Agents

Hierarchy-based agents not only concern themselves with cognitive aspects such as knowledge or goals but also with architectural elements such as mobility, distribution, dynamic adaptability or hierarchical representation of agents. They do this by combining various reasoning hierarchies and distribution. This allows operators to manipulate the hierarchy or define how beliefs are shared. Authority, parent, knowledge, goals, messages, capabilities, process and sub-agents define agents and their classes. The creating agents set the authority of that agent represented hierarchically. Information about other agents or about their environment is the knowledge the agent possesses.

An agent tries to achieve it goals by utilizing its own capabilities, or else by availing itself of services that other agents offer. Agents communicate using messages, and take necessary actions to complete goals. Agents have the ability to execute multiple processes and they have a list of all sub-agents. An agent can exhibit reactive or proactive behaviors. It can be reactive by taking a message and executing all applicable capabilities or it can be proactive by picking a goal, finding applicable capabilities, and applying them to achieve an outcome.

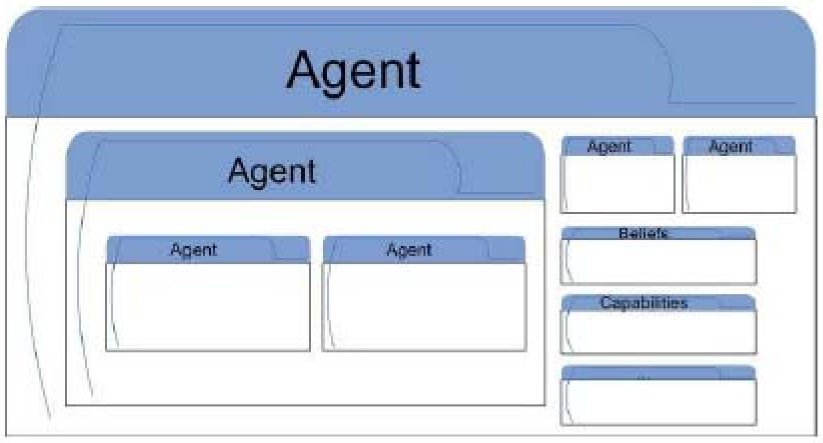


Figure 6. Hierarchical Agent (From [1]).

6. Task and Communication Agents

These agents do not provide dedicated constructs to represent mental categories or intelligent behaviors. This Agent type tends to be a social piece of object-oriented software without a deliberation mechanism. The software that best describes the development of these agents is JADE (Java Agent DEvelopment Framework). JADE consists of an in-depth library of behaviors that the programmer utilizes to program the activities of the agent [30].

There are multiple kinds of behaviors, for example, cyclic or parallel. There is a queue of active behaviors for each agent and the agent is concerned with how these behaviors are scheduled. The agent executes its behaviors by using a non-preemptive, round-robin schedule.

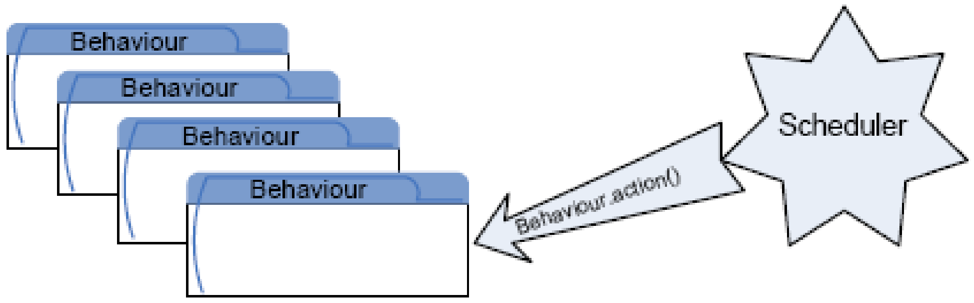


Figure 7. Task-Based Agent (From [1]).

D. SUMMARY

A taxonomy for agent types has been discussed six agent types are identified and their main features are emphasized. The taxonomy shows their advantages and drawbacks. Different approaches for programming agents are presented, issues they handle well and not were highlighted. No agent type is better than another. Users need to familiarize themselves with different types and pick the most suitable one for what they are trying to accomplish. Knowing more about each agent will improve one’s experience.

Agent	Deliberative Control Flow	Mobility	Reactivity	Proactivity	Capabilities	Belief
Commitment	Rules to establish commitment	No	Yes	Yes	Abilities of the agent under certain conditions	Time-stamped facts
Event-driven	Plans that handle relevant event	No	Yes	Yes	Reusable module formed by plans, beliefs	Frames (open and closed-world
Goal-directed	Practical reasoning rules to satisfy goals	No	Yes	Yes	Abilities of the agent under certain conditions	Prolog program
Software integration	Agent program with some logic	Yes	Yes	Yes	No	Interfaced-legacy code
Hierarchical	Capabilities or plans	Yes (Java)	Yes	Yes	Abilities of the agent under certain conditions	Frames
Task-based	Non-preemptive round robin schedule for tasks. No deliberation	Yes (Java)	No	No	Tasks	No (only JAVA objects

Table 1. Comparison of the main computational aspects and main mental categories of the six classes of agents (From 1).

III. SURVEY OF SELECTED AGENT-BASED SOFTWARE

There are well over 100 ABMS software systems available to users. Our objective is to identify a subset of the most popular software platforms and the following attributes for each system:

1. agent structure(s) which each system employs;
2. technique(s) for representing agent behavior which each system employs (e.g., state transition graphs, programming language procedures, rules, etc.);
3. where in the agent taxonomy presented in the previous chapter, each system fits, noting that any single system may inhabit several of these classes;
4. other salient features such as cost, host programming language, operating system(s), Application Programming Interface (API), open source / proprietary, and level of technical expertise required to use the system.

The six systems we examine in detail are NetLogo (Northwestern University), SWARM (Santa Fe Institute), AnyLogic (Xytec, Inc.), REPAST (University of Chicago), MASON (George Mason University) and Ascape (Brooking Institute). It is interesting to note that most of these systems have been developed at research-oriented institutions, which leads us to believe that this decision technology has not yet evolved for use by a wider, less technically sophisticated audience.

A. TAXONOMY CLASS(ES) OF SELECTED SOFTWARE SYSTEMS

1. NetLogo (Multi-Platform, Freeware)

Based on StartLogo, NetLogo was created by Uri Wilensky in 1999 [26]. It is a cross-platform, multi-agent general-purpose complexity programmable modeling and simulation environment. It is in continuous development at the Center for Connected Learning (CCL) and Computer-Based Modeling at Northwestern University. NetLogo is designed to model complex systems as they evolve over time. Modelers can give multiple instructions to hundreds of independent agents in a simulation environment. Through the interactions of these agents, one can explore the connection between the micro-level behaviors of individuals and the macro-level patterns that develop from their interactions.

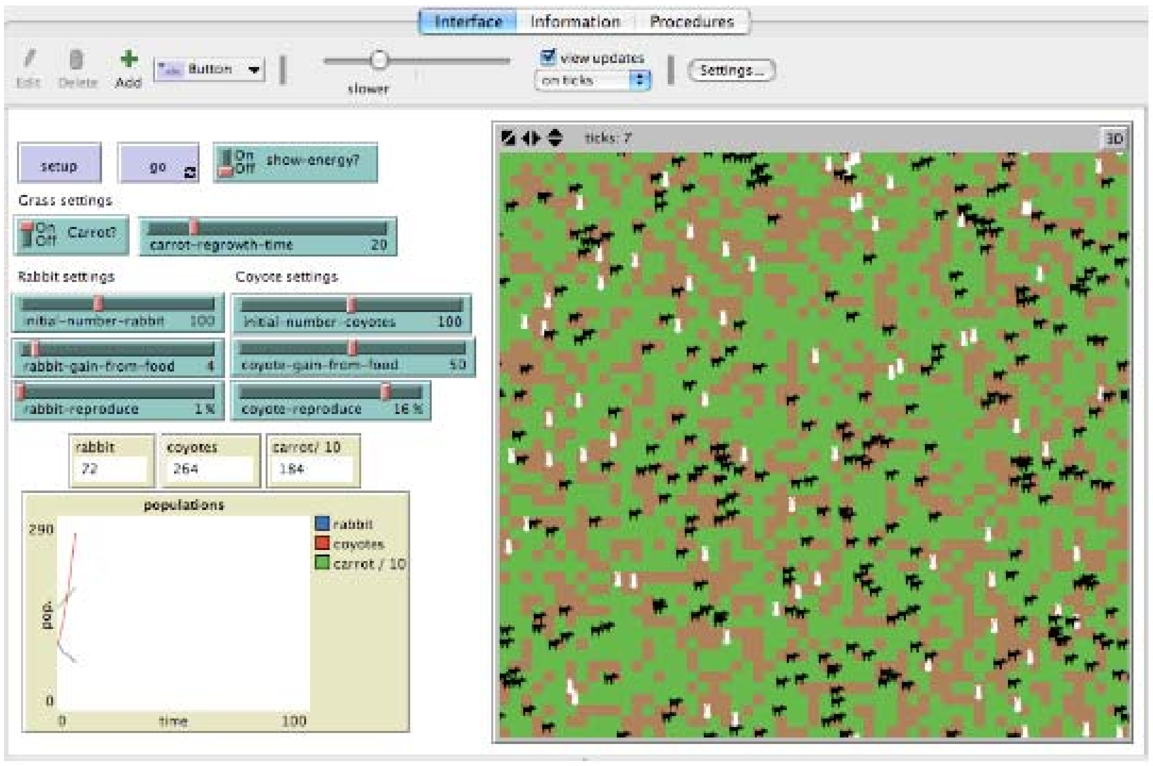
NetLogo is simple enough to allow users to open existing models or create one’s own simulation and advanced enough for researchers to use in the field. Through the use of HunNet, a network of computers or handheld devices, modelers can control individual agents in a simulation. NetLogo claims to be the next generation of multi-agent modeling languages. It is implemented in Java and it will operate on all platforms, including Mac. It runs as a standalone application and can be implemented as Java applets inside a browser.

In NetLogo, the world (environment) is a two dimensional grid of “patches” (individual squares in the grid). This view can be changed to 3D with a click of a button. There is a library of reusable sample models that can be modified to fit specific modeling needs. In any sample model, there are sliders and switches to modify agents. For example, one can specify graphically using slide bars simulation parameters such as the initial number of rabbits and coyotes, the carrots’ growth rate, and the speed of the simulation.

Creating an executable model is easy with little programming ability and simple codes. NetLogo provides many sample models to learn from. People with little programming experience can choose a model from the models library to get started. Once the model is opened, it can be manipulated by adjusting the interface’s sliders and switches. To add new parameters controls, simply add it to the interface and assigned it to a variable. Variables must be defined in the programming codes using the procedures tab, which creates the environment and agents. Commands (rules) are instructions given to agents that set their behaviors.

In the ecosystem example, rabbits, coyotes and carrots have rules. Rabbits must eat carrots in order to survive and, in turn, coyotes must eat rabbits to survive. All agents get energy and if their energy runs out, they become slow (i.e., “low energy” rabbits are more easily caught; “low energy” coyotes catch fewer rabbits). Reproduction rules can also be added to continue the simulation. Figure 8 is a modified wolf-sheep predation sample. The wolf (wolves) was modified to represent coyote (coyotes), sheep to rabbit and grass to carrot.

In the sample model, additional attributes (energy, reproduction, etc) have been added. To modify the sample further (adding more variables and commands) requires programming expertise. Adding sliders, buttons, monitors and even creating shapes and modifying the environment are easy to accomplish in NetLogo. (See the Appendix for samples of code which instantiate this simulation.)



show-energy? Off
Grass settings
Carrot? On
carrot-regrowth-time 20
Rabbit settings
initial-number-rabbit 100
rabbit-gain-from-food 4
rabbit-reproduce 1 %
Coyote settings
initial-number-coyotes 100
coyote-gain-from-food 50
coyote-reproduce 16 %
rabbit 72
coyotes 264
carrot/ 10 184

Figure 8.   NetLogo screen shot of coyote rabbit predation model.

NetLogo can be downloaded at http://ccl.northwestern.edu/netlogo/ (Jan2008) as freeware. The website provides extensive documentation and tutorials. There is also a Models Library, which consist of pre-written simulations by other users. The available models can be modified to your specification. These simulations address many domains including biology, physics, chemistry, mathematics, economics, social psychology, and computer science.

2. SWARM Simulation System (Objective C and JAVA, Open Source)

Swarm was developed by researchers at Santa Fe Institute for building discrete event simulations of complex systems with heterogeneous elements (agents). Swarm started as a general language and toolbox for ABMS and was intended for use across a wide spectrum of scientific domains such as biology, physics and social science. The software implements both a model and a separate virtual laboratory for observing and conducting experiments on that model.

The basic, underlying concept is to design a model based upon a hierarchy of “swarms,” each “swarm” being a group of objects and a schedule of actions that is executed by the objects. An agent is a basic unit of a swarm, an entity that generates events that affect others and/or itself. For example, in an ecosystem simulation environment, coyotes, rabbits and carrots represent agents. Agent behaviors are represented as discrete events such as “rabbits eat carrots”, “rabbits hide from coyotes” and “coyotes eat rabbits”.

Many different environments can be modeled in Swarm including stock markets, shopping malls, parking lots, buildings, and rain forests with corresponding agents such as bankers, shoppers, drivers, customers, and animals. Swarm consists of a collection of software libraries, implemented in Objective C, an object oriented (OO) language. This language was chosen because it lacks strong typing, which is in keeping with the complex-systems philosophy of de-emphasizing centralized control, and because it can execute faster than Objective-C.

It should be noted that the scalability of agent systems is an important issue since the computation time for agent-based simulations typically rises exponentially with the number of agents. Definitions of various classes of objects are represented in OO programming software. An object’s state and associated methods that implement its behavior consist of a combination of instance variables of that object.

An agent is modeled directly as an object in Swarm. The classes represent the types of agents (e.g., generic coyotes) and objects (agents) are instances of the classes (e.g., a particular coyote). Individual objects have their own state variables, but the class defines the generic definition of its behavior. Particular characteristics are stored as variables, or properties, in the class. The methods implement the behavior(s) of the agent. The schedule is an ordered series of actions to be performed by objects, sometimes based upon existing preconditions of the landscape.

There are seven core libraries in Swarm, four of which (defobj, collections, random and tkobjc) are support libraries for use outside of Swarm, and the remaining three (activity, swarmobject and simtools) which are for internal Swarm use only. Currently there are three (space, ga and neuro) domain-specific libraries for Swarm model builders. These libraries contain the classes needed to create agents. In the ecosystem example, many simulation platforms will fix the environment as a two-dimensional grid.

In Swarm, the environment is also an agent because it has no design requirement for a particular environment. So, the first object class in this prey-predator example is the environment itself (360 ft by 360 ft) followed by Coyote, Rabbit and Carrot object classes. Individual Coyote, Rabbit and Carrot objects can be tailored by adding additional methods. Specific methods are defined to refine behaviors of specific agents, for example, rabbits may be subtyped as Large, Medium, or Small with associated behaviors that Large Rabbits will eat carrots 6 hours a day while Small Rabbits will eat 4 hours a day.

Once the agents are defined and relationships are established, the agents need to be put into a swarm. The user then writes a schedule of activity, defining the time dimension of the simulation. Users build schedules by creating instances of data structures from the activity library, putting together ordered object/message pairs.

A model can tell objects to execute some action(s) without knowing what types of objects are on the list. Swarm represents model objects by using its own data structures and memory management. Swarm implements tools called "probes" that allow users to monitor and control any simulation object from the graphical interface or within the code. A Swarm model usually consists of 4 components:

(1) an observer: provides a graphical depiction, graphs, control panels and parameter display of the model;

(2) a model: creates the agents, objects and schedule;

(3) a collection of agents; and

(4) an agent's environment.

Swarm provides explicit methods for scheduling, which may be either dynamic or static. Swarm has also been implemented in Java, because of strong demands from its user community to coordinate message passing from Java's strong typing to the Objective-C library.

Swarm (Objective-C and Java) can be downloaded at http://www.swarm.org (Jan2008). In addition, a user guide with tutorials is available. The website also has a contributed-code section where users can provide software for re-use. This reusable code includes classes written for:

* Genetic algorithms
* Neural networks
* 3-dimensional spaces
* Boolean networks
* Output files, including summary statistics on agent lists
* Date and time management

3. AnyLogic: Multi-Paradigm Simulation Software (Java, Proprietary)

AnyLogic is a simulation tool use to build highly sophisticated web-based simulation models that support process-centric (discrete event), system dynamic and agent-based modeling approaches. AnyLogic can be applied to many different domains to build models for business, strategy, economics, social systems, war-gaming, biological systems, physics and software performance. AnyLogic is not based upon any single classical simulation modeling paradigm, but rather supports multiple simulation paradigms using approaches and languages for handling complexity that have been adopted from the software engineering world.

The object-oriented core language of AnyLogic supports system dynamics-based stock-and-flow diagrams and discrete event-based simulations in addition to agent-based simulations. The ability to rapidly compose and integrate industrial-strength agent-based models within the same visual environment is a powerful feature, which distinguishes AnyLogic from the other software platforms. In addition to rich visualization capabilities, AnyLogic supports ready-to-use constructs for defining agent behavior, communication and environment models. It provides a more versatile capability for modeling large, complex systems as a result of being able to combine different paradigms.

AnyLogic supports virtually all approaches to discrete event and continuous modeling like flow diagrams, system dynamics, agent-based modeling, state charts, etc. It uses an open architecture and interoperates with any office or corporate software written in Java or other languages. Data can be read dynamically, exported to spreadsheets and databases, or be embedded in real-time operational environments. External programs can utilize any of the simulation models using the open API and can be called from anywhere in the model. There are also random number generators, numerical methods and optimization algorithms available for building both stochastic and deterministic models.

AnyLogic provides a powerful capability for representing complex dynamic behaviors called statecharts. It enables a graphical view of different states of agents, their transition, events that trigger them, timing and agent’s actions in its lifetime. Statecharts provides a more general representation medium that is superior to the code-oriented representation used by Swarm and other systems below. With Statechart, it is easy to see the level of description of each state the agent is in. Several aspects of an agent’s life (family, work, etc.) can be analyzed by having multiple statecharts working in parallel and interacting.

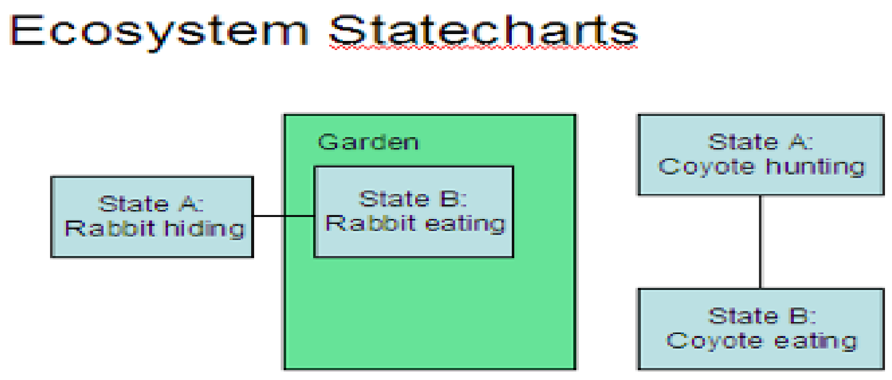


Figure 9. Ecosystem Statecharts.

In the above statechart, rabbits and coyotes both have two states. Rabbits are either hiding or eating, and coyotes are either hunting or eating. This is a very simple example of a statechart. In AnyLogic, the environment will consist of a 2D grid, and behaviors of agents can be further specified. For example, rabbits will continue to eat until the carrot supply is zero, or a coyote will die if it does not eat a rabbit within two days, and so on. A cyclic timer can be created to help determine behaviors that are time dependent (life, schedule, etc). In AnyLogic, a 2D representation allows for easy viewing and the ability to trace a single agent lifespan throughout the simulation.

AnyLogic is implemented using Java. The website (http://www.coensys.com/index.htm (Jan2008)) provides a fully functional 15 days evaluation version. The chart below shows the cost breakdown for AnyLogic 6.

License Type	Price per	w/ OptQuest	Maintenance fee (Annual)
Advance Edition (1)	6199	7299	2200
(2)	5100	5850	1800
(3)	4250	4950	150
(4+)	3950	4550	1400
Professional Edition (1)	15500	N/A	4900
(2)	12800	N/A	4050
(3)	11199	N/A	3500
(4+)	10199	N/A	3200
Single Educational	$395	490	$145
Faculty/Department	$690	850	$250
Faculty/Department w/ Home Usage	$990	1400	$250

Table 2. AnyLogic cost chart.

4. REPAST (Java, Python, C#, Open Source)

Repast (Recursive Porous Agent Simulation Toolkit) was designed for building agent-based models and simulations in the field of social science. Researchers at the University of Chicago and the Argonne National Laboratory originally created Repast to implement Swarm, or equivalent functionality, in Java. Because Repast does not adopt all of Swarm’s design philosophy, it does not constitute a full implementation of Swarm. In addition to most of Swarm’s features, Repast has the capabilities to reset and restart models from the graphical interface and the “multi-run” experiment manager. Its execution speed is much faster and it consists of many classes for geographical and network functions.

Repast makes it easy for inexperienced users to build models by including a built-in simple model and provide interfaces through which menus and Python code can be used to begin model construction. Repast offers an appealing feature in its ability to integrate geographical information science (GIS) data directly into simulations. It is now being managed by a non-profit organization called ROAD (Repast Organization for Architecture and Development). Repast comes in four versions supporting three languages: RepastJ (Java); RepastPy (Python); Repast.Net (C#, any .Net language); and RepastS (Simphony, Java-based). It will run on virtually any modern platform including Mac.

Repast is a complex program and one needs to have experience in programming codes in order to use it. There are model templates available for reuse and modification. A model is created using codes like setup(), begin(), buildModel(), etc. The model display is 2D. The rules and schedules must be incorporated in the codes. Lists of the varying parameters (NumRabbits, RabbitType, etc) and 'get' and 'set' methods for each parameter in the list must be supplied. Additionally, the environment (currentSpace) parameters have to be declared in the codes. The agentGrid holds the position of each agent in the simulation and provides a way to locate an agent without searching through the list of all agents. This software interface is not amenable to users without programming experience.

Repast can be downloaded at http://repast.sourceforge.net/download.html (Jan2008) for free and for multiple platforms. A self-study guide, prepared by Leigh Tesfatsion, can be obtained from http://www.econ.iastate.edu/tesfatsi/repastsg.htm (Jan2008) to help newcomers get started programming with RepastJ.

5. MASON: Multi-Agent Modeling Language (Swarm Extension)

The George Mason University's Evolutionary Computation Laboratory (ECLab) and the Center for Social Complexity developed MASON to serve as a basis for multi-agent simulation tasks ranging from swarm robotics to machine learning to social environment. The foundation for MASON's design ranges from large custom-purpose to lightweight simulation needs. It is implemented in Java and provides a fast discrete-event multi-agent simulation core and visualization toolkit.

MASON delineates between model and visualization. This allows models to be dynamically detached from, or attached to, visualizers allowing simulations to change platform midrun. MASON is designed for use in a wide range of simulations with a special emphasis on “swarm” simulations involving millions of agents.

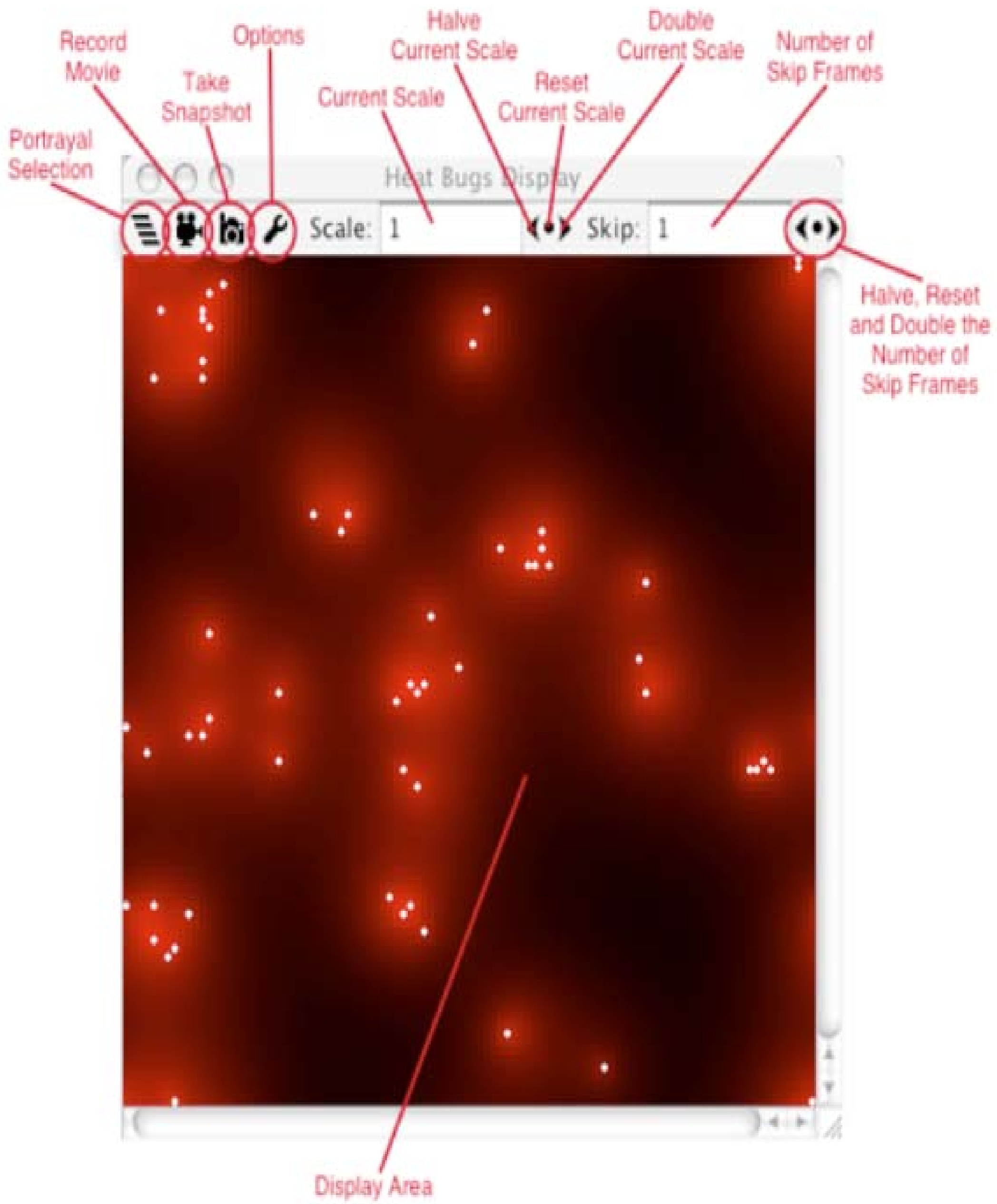


Figure 10. 2D visualization by Mason (From 18).

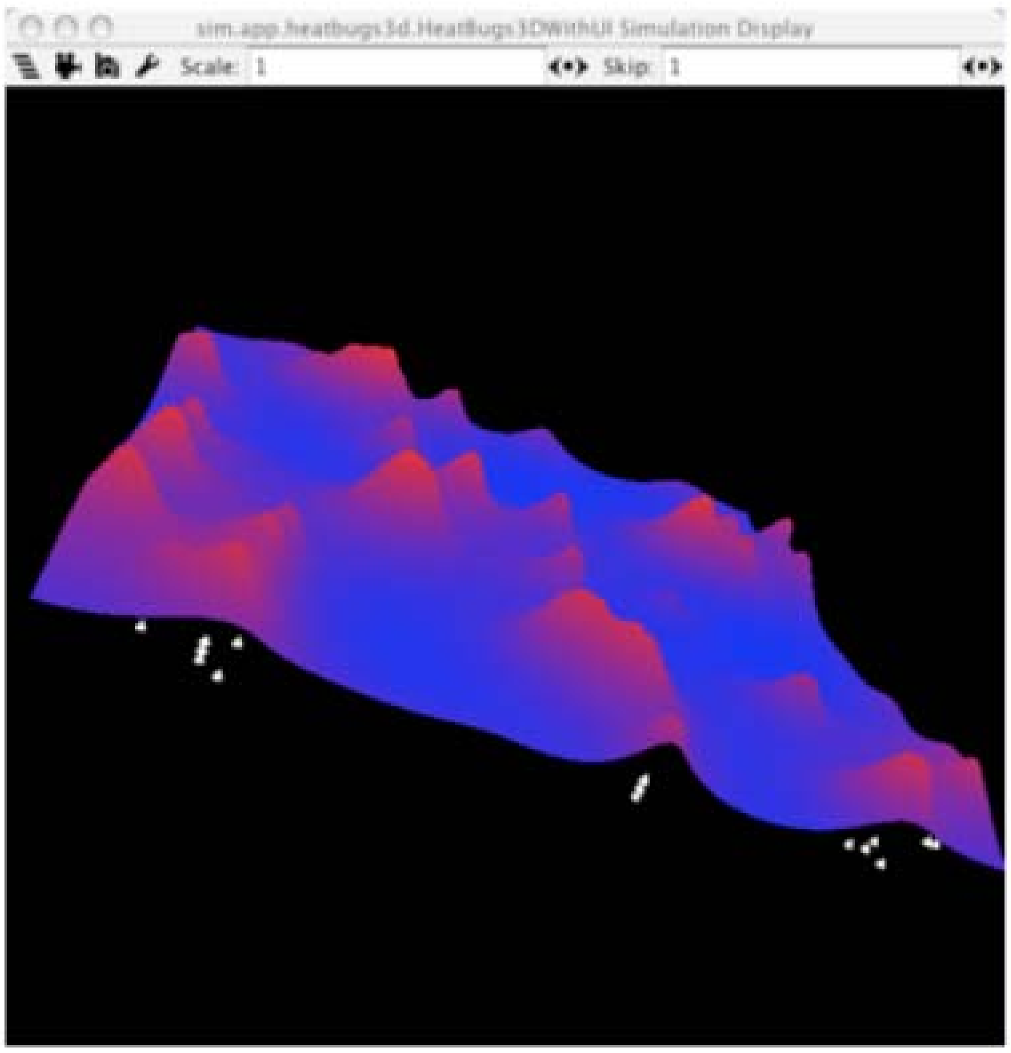


Figure 11. 3D visualization by Mason (From 18).

MASON was developed from scratch and was designed to be a fast, minimal model library with an optional suite of visualization tools in 2D and 3D that can produce screenshots and movies. The 3D capability is an enhancement from Swarm because it allows for photo and movies features. The library was designed to add features in a modular fashion as opposed to many systems, which tend to be domain-specific and have hard-wired features that are difficult to remove and modify. In addition, GUI facilities were added because it was found useful for a variety of simulation tasks.

Researchers use the library to perform many simulation runs with large numbers of agents and interactions, and with visualization and modification of the runs. MASON’s objective is to provide a fast, portable, capable means of guaranteed-duplicated results independent of platform by check-pointing and restarting models with or without visualization. These models also have the ability to migrate across platforms. The major differences between Swarm and mason are the scalability and Mason visualization and GUI tools are separate from the model allowing photo and movies capabilities.

Creating the ecosystem requires coding experience. The environment and behaviors (rules) are defined in the codes. It is much easier to understand the communication among objects with Mason over Swarm. The model’s ‘get’ method (inside it Step method) allows it to communicate with other by holding information needed by other objects. Mason is good for experienced programmers doing computationally intensive (many agents or long run times) models because it tends to run faster.

MASON is a free download at http://cs.gmu.edu/~eclab/projects/mason/(Jan2008). Optional libraries can be downloaded to generate movies, charts/graph or recompile MASON. Various examples are also provided to help users visualize the outcomes of their models.

6. Ascape (Agent Landscape)

Ascape was developed at Brookings Institute as an agent-based environment that is not only easy to use, express, flexible and powerful but also accessible to a wider array of users beyond the programmer community. The framework supports complex and sophisticated agent-based models. Non-programmers can explore various aspects of model dynamics by utilizing the end-user tools. Models developed by Ascape can be simply published to the Web for use with common Web browsers.

Two fundamental concepts underlie the Ascape framework: agents and “scapes” (territories on which agents live). Scape, in Ascape, provides a way to organize agents into a collection because the behavior of a collection of agents is of paramount importance in agent-based modeling. This idea simplifies the execution of model behaviors, the collection of simulation statistics, and the composability of models. A scape is also an agent, so that scape hierarchies can be built which allow models to be embedded in one another. This composability feature facilitates the modular construction of simulations that can then be linked with one another to build more complex models.

Ascape is implemented in Java and the models are represented as JAVA classes. The framework is amenable to all levels of programmers, from novice to expert. Novices can utilize pre-built example models or models built by other users. Models can be manipulated by interactively adjusting the rules through the Model Setting windows of the graphical user interface (GUI). In addition, users can also change the parameter values, assemble charts and capture videos of models for later viewing. Users can then export data to Microsoft Excel.

Ascape models can also be executed on the web as applets. In addition to a library of commonly used rules (walk, die, eat, etc), Ascape also has a built-in “statistic collector” that automatically computes various functions like count, sum, standard deviation, etc. Ascape does not limit expert programmers even though it provides many built-in features. Its general-purpose language framework (Java) allows programmers wide flexibility.

Ascape reduces a programmer’s chance of making mistakes by allowing him/her to concentrate on the models rather than the simulation infrastructure. Ascape separates implementing code from viewing and controlling code. Ascape provides a feature-rich skeleton application from which to start model construction. Developers can often create an executable application with multiple features by just adding a few lines of code to specify parameters and methods. The createScape method creates agents, which might be scapes themselves. With very little experience, a beginner can study the example model code provided and make rapid progress.

With little knowledge of Java, a user can create the ecosystem example by utilizing a sample model and modifying it with the many common rules (behaviors) provided by Ascape. With so many features and sample models, it will not take long for a beginner to start building their own models. It is free for non-commercial use and a tutorial by the developer (Miles Parker) on design, development and use of Ascape can be downloaded at: http://ideas.repec.org/a/jas/jasssj/2000-13-1.html (Feb 2008).

B. SUMMARY OF AGENT SOFTWARE PLATFORMS

Tables 3 and 4 show a summary of the six software systems we reviewed above. Some of the main observations we draw from this table are:

1. A significant level of software coding knowledge is required for the construction and execution of agent-based models using these systems. Even for NetLogo and Ascape, where relatively little programming expertise is required to get started, nontrivial coding skills are necessary to create models that are more complex.
2. There are very limited higher-level agent representation schemas. Only AnyLogic provides any capability in this realm in the form of statecharts for capturing agent behaviors.
3. Most systems provide some degree of reusability of models in the form of model libraries, templates from which users can retrieve and subsequently modify.
4. Scalability in terms of the number of agents that the system can support is high for Mason, AnyLogic and Ascape while medium for the remaining systems.

	NetLogo	Swarm	Anylogic	Repast	Mason	Ascape
Agent structure representation	Code	Codes (Swarm)	Codes (Stock-and-flow, Flow chart	Code	Code	Code
Agent behavior representation	Code (command and procedures)	Codes (Methods	State transition diagrams (Rules)	Code (Rules)	Code (Rules)	Code (rules)
Host Languages	Java	Objective C and Java	Java	Java	Java	Java
Open source?	Yes	Yes	Yes	Yes	Yes	Yes
Language(s) and API	Java API	Objective C and Java API	Java API	Java API	Java API	Java API
Operating system(s)	Windows, Mac and Sun Java	Windows, Unix, Linux and Mac	Windows, Mac and Linux	Windows, Mac and Eclipse	Windows, Mac and Linux	Eclipse and independent

Table 3. Software features.

	Swarm	Anylogic	NetLogo	Repast	Mason	Ascape
Composability	Yes	Yes	Yes	Yes	Yes	Yes
Scalability	Medium	High	Medium	Medium	High	High
Excel Export	No	Yes	Yes	Yes	No	Yes
Models Librar	Yes (7)	Yes	Yes	Yes	Yes	Yes
Programming Experience	High	High	Low	High	High	Low

Table 4. Software features (cont).

C. RELATIONSHIP BETWEEN AGENT TAXONOMY AND SOFTWARE SYSTEMS.

Matching the agent systems with the agent taxonomy requires some level of arbitrary assignment since each software system can be seen as embodying some aspects of the various agent types discussed in the previous chapter. However, when focusing upon the primary agent objective in each case, we reach the following classification as shown in Table 4:

(1) NetLogo implements mainly Hierarchy-based and Goal-directed agents,

(2) Swarm emphasizes Commitment and Hierarchy-based agents,

(3) AnyLogic embodies Event-driven and Software-integration agents,

(4) Repast reflects not only Hierarchy-based and Commitment but also Software-integration agents,

(5) Mason employs Task-, Communication- and Software-integration agents,

(6) Ascape emphasizes Goal-directed and Commitment agents.

	NetLogo	Swarm	AnyLogic	Repast	Mason	Ascape
Commitment		X		X		X
Event-driven			X			
Goal-directed	X					X
Software-inegration			X	X	X	
Hierarchy-based	X	X		X		
Task-, Communication-					X	

Table 5. Software and type of agents (taxonomy) relationship.

D. SUMMARY

The most popular agent-based modeling platforms appear to require substantial software engineering expertise in order to build and analyze models. This artificially restricts the community of potential users, and limits the accessibility of this decision technology. In the next chapter, we develop a conceptual architectural framework for an agent-based modeling environment which relaxes this constraint.

IV. CONCEPTUAL FRAMEWORK FOR GENERALIZED AGENT-BASED MODELING ENVIRONMENT (GAME)

The summary of agent-based software conducted in the previous chapter reveals that modeling software in this domain is at a relatively immature level, similar in many ways to where modeling software for operations research and management science (OR/MS) applications were two or three decades ago. Developments such as modeling languages resulted in software which opened its usage to a broader, less mathematically inclined audience, including decision makers. We would like to replicate the success of software evolution in the domain of OR/MS to the area of agent-based modeling environments. In this vein, we develop a broad conceptual architecture which incorporates some of the design advances realized in OR/MS modeling software. Our intention is that this technology transfer can be used to design generalized agent-based modeling environments (GAMEs) that will reach a wider community of users than is currently the case.

A. DESIGN REQUIREMENTS

We present a list of high level requirements culled from the shortcomings discovered in our survey in C3, and from model management design principles as articulated, for example, in [29]:

a. Interfaces that serve the entire spectrum of potential users: analyst, decision-maker and programmer. GAME software must accommodate all users by providing not only data collection capabilities but also analysis tools and general representation frameworks for models and data. Any coding process should be as transparent to the user as possible; in this sense a GAME environment could ideally be thought of as a CASE tool for agent-based models. If customized coding is required, interfaces for programmers should be straightforward not only for programmers but also accessible to nonprogrammers if at all possible. The ability to import and export data to spreadsheets, databases or other systems is also highly desirable.

b. Higher-level agent model representation techniques. Statecharts, for example, provide a graphical view of the agent different states, their transitions, causes of transitions, timing, and agent’s actions during its lifetime. This representation technique enables users to have a higher-level understanding of the agent’s more complex dynamic behaviors without being a software engineer. Looking at a typical statechart, it is relatively easy to see the different levels of description agents can have. Statecharts provide one methodology by which complex real time systems can be represented in an intuitive graphical manner. Not only do they enable complex relationships between concurrent states to be formed through synchronization techniques but also though decomposition of states. It should be noted that statecharts are only one possible representation technique for agent behaviors. One of the desiderata for a GAME should be support for multiple high level representations, including the conversion capabilities for mapping these representations to available solvers.

c. Separation of agent model representation from the solver code. In the OR/MS world, models are represented in entity-relationship form, and then tied to solvers dynamically at run time. We would like to implement a similar scheme in the agent-based environment where the solvers are the software systems we have profiled in the previous chapter (NetLogo, Ascape, etc). Another interesting research direction in this area is to explore if, and how well, spreadsheets can serve as solvers for agent-based models.

d. Transformation procedures for converting agent model representations (in b.) to solver code (in c.). For example, a user may want to represent agents as state transition diagrams and then use NetLogo to run the execution. In general, users should be able to select which solver they want to apply to an agent model with a click of a button. This flexibility requires the development of preprocessors which can map agent representations into object-oriented code. Such an architecture would enhance a user’s chances of matching a simulation with the solver best suited for its execution. For example, a large-scale simulation with thousands of agents might exceed NetLogo’s capacity, and require a more scalable solver such as MASON.

e. Model reusability via libraries of existing models. The libraries must be easy enough for nonprogrammers to access and understand. Ideally, it would be desirable for users to simply drag and drop existing models into their workspace and then modify the representations and original codes through a simplified interface. Additional model documentation may be necessary in order to facilitate this kind of agent behavior reuse.

f. Language or equivalent interface for experimental design. Simulations typically consist of models plus experimental designs for running and testing the models. For example, a user may want to run a whole series of Coyote_Rabbit simulations (experiments) where s/he varies the initial ratio of the population of Coyotes to Rabbits from 0.01 to 0.2 by increments of 0.1, and compares the final ratio of Coyotes to Rabbits, plotting the results for each simulation. A notional language representation for such an experimental scenario might be accomplished as follows:

```
RUN COYOTE_RABBIT MODEL
WHERE
INIT_POP(RABBITS) = 1000
INIT_POP_RATIO =
INIT_POP(COYOTE) / INIT_POP(RABBITS)
VARYING INIT_POP_RATIO FROM 0.01 TO 0.20 BY 0.01
SAVING FINAL_POP_RATIO = FINAL_POP(COYOTE) /
FINAL_POP(RABBIT)
```

```
ITERATIONS = 250
PLOT FINAL_POP_RATIO VS. INITIAL_POP_RATIO
USING NETLOGO
END RUN
```

Again, this language should be available not only in “native”, or programming mode as above, but also augmented by graphical means so that it can be easily adopted by nonprogrammers.

g. Ability to link with models from other paradigms. The real power of a generalized modeling environment is to bring together models as they are needed independent of the “type” of model. The ability of agent-based models to “play” with other models can be very useful. For example, we may want to use a simple optimization model as a decision model for specifying our agents’ behaviors in a resource allocation environment. Conversely, we may want to develop an agent-based model for exploring efficiently the solution space of a complex, parameterized optimization model to find global optima.

B. CONCEPTUAL ARCHITECTURE

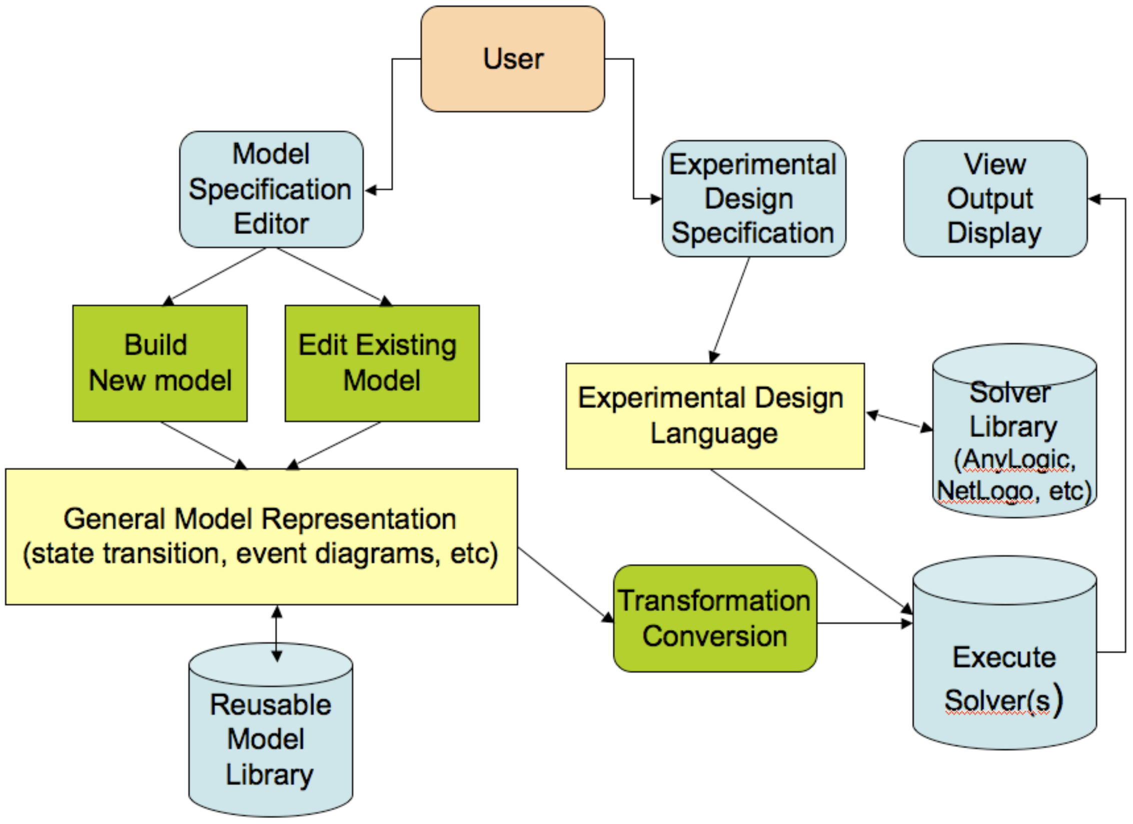

User
Model Specification Editor
Experimental Design Specification
View Output Display
Build New model
Edit Existing Model
Experimental Design Language
Solver Library (AnyLogic, NetLogo, etc)
General Model Representation (state transition, event diagrams, etc)
Transformation Conversion
Execute Solver(s)
Reusable Model Library

Figure 12. Design requirements diagram.

Figure 12 shows a conceptual architecture which captures the design desiderata outlined above. In this architectural environment, a user can do four things: create a model representation either from scratch or by selecting or modifying a model from the library, generate an experimental design using an experimental design language (EDL), solve the model using the environment's transformation preprocessing routines to convert the model representation into the appropriate code for the solver specified by the user in the EDL, and display the results of the simulation.

The main components of the architecture to facilitate this process flow are a model specification editor for representing models in generalized formalisms (e.g., statecharts), a library of reusable models, an experimental design language for specifying the initial conditions, the solver to be used and the data collection to be done, a library of solvers (Ascape, NetLogo, AnyLogic, etc), a library of transformation preprocessors, and visualization software for viewing the results.

What we have not shown in this framework is any significant database management presence. Agent-based models tend not to rely so heavily upon existing database resources as traditional analytical models do (e.g., optimization, regression). Thus, we assume that data management tasks can be handled locally without having to specify formal data representation techniques. This assumption may be naïve however and it would be an interesting research issue to explore when such capabilities are needed and how they would be integrated into the environment.

	NetLogo	**Swarm**	**AnyLogic**	**Repast**	**Mason**	**Ascape**
Interfaces	Yes	Yes	Yes	Yes	Yes	Yes
Model Representation			Yes			
Solver Code	Yes	Yes	Yes	Yes	Yes	Yes
Transformation						
Libraries	Yes	Yes	Yes	Yes	Yes	Yes
Language/experimental design	Medium	Low	High	Medium	Low	Medium

Table 6. Reviewed Software and conceptual architecture requirement design relationship.

Table 6 shows the relationship between the conceptual architecture requirement design and the six reviewed software systems for agent-based modeling. Most of the software systems do not have model representation but are potential solvers. AnyLogic appears to be the closest system to satisfying the architecture in Figure 11 although it, like all the other systems, has no transformation capability to facilitate cross–platform work. AnyLogic also appears to be the only system which supports anything approaching the multi-paradigm modeling feature which allows integration of models from different reference disciplines. AnyLogic supports discrete event, agent-based, and system dynamics models within its environment. Thus, although none of the systems meets the complex demands of a GAME, AnyLogic appears to provide the best template for considering how to eventually implement such a modeling environment.

C. SUMMARY

Due to the relative immaturity of ABM software, this decision technology is not as accessible to potential users as we would like. The need to have programmers intricately involved in model specification and implementation significantly and needlessly limits the usability of this decision technology. Our intent is to create a conceptual architectural design whose implementation would overcome some or all of these obstacles. Accordingly we have developed requirements for interfaces, which would serve a much wider range of stakeholders.

This architecture relies upon a higher-level model representation separate from the solver code, and a library of transformation procedures to map from representations to solvers. Further, extensive, more richly documented reusable model libraries would provide templates simple enough for all users, from novice to experienced. A robust language or equivalent interface for specifying experimental design procedures would also amplify the value of such software. With rapid technology growth and the spreading popularity of ABM, there is promise that ABM software platforms can be improved to extend beyond just the research community and make its way into the hands of real decision-makers.

THIS PAGE INTENTIONALLY LEFT BLANK

V. SUMMARY

Decision support technologies, much like databases in the past, stand in relative isolation from one another. The ability to access and integrate a wide range of such technologies in an IDTE setting can potentially increase their utility as well as realize economies of scale in building more complex decision-oriented systems.

Agent-based modeling and simulation is a rapidly rising decision technology which, to date, has remained in the bailiwick of programmers and technical analysts. The objective of this research has been to begin identifying ways in which to overcome this artificial limitation, by leveraging advances in modeling software from the OR/MS domain. Specifically, we have set out to investigate design requirements for a generalized agent-based modeling environment (GAME) that transcends the current state of the art in agent-based software platforms.

In the process of identifying GAME requirements, we first adopted a taxonomy of agent types, based upon their properties and roles. Six agent types were identified: commitment, event-driven, goal-directed, software-integration, hierarchy-based and task-, communication-agents. Six popular ABMS software systems were then reviewed and the relationships with agent types were compared. Most software systems could be mapped to two or three agent types. Every software system offers libraries of model examples, some of which can be relatively easily modified and adjust, and others that require high levels of coding experience.

The thesis clearly showed that most popular agent-based modeling systems require a significant degree of software engineering expertise to use them effectively. The maturity of ABMS software is quite low, and comparable to the state of modeling software in the fields of OR/MS two decades ago. Experience with OR/MS modeling advances indicates design requirements for a more generalized environment, which we incorporated into a high level conceptual architecture for a GAME. This architecture has as its major components a high level model representation schema, an experimental design language (EDL) for specifying simulation runs, a library of reusable models, a library of solvers in the form of existing software platforms such as the ones reviewed, and a library of conversion routines for mapping model representations into solvers.

This architecture presents exciting opportunities for model software development and advancing the discipline of agent-based modeling and simulation. Comparable improvements in GAME implementation to other modeling environment advances can eventually result in agent-based software that will serve a much wider range of stakeholders, as well as integrate with other decision technologies in an IDTE to address more complex decision-making challenges.

APPENDIX

Below are the NetLogo codes for the Coyote Rabbit ecosystem sample. Everything that follows “;;” is in the nature of a comment.

```
Setup codes to establish agents:

;; rabbit and coyotes are both breeds of turtle.

       breed [rabbit a-rabbit]  ;; rabbit is its own plural, so we use "a-rabbit" as the
       singular.

breed [coyotes coyote]

turtles-own [energy]       ;; both coyotes and rabbit have energy

patches-own [countdown]

to setup

clear-all

ask patches [ set pcolor green ]
```

```
Carrot codes:

;; check carrot? switch.

       ;; if it is true, then carrot grows and the rabbit eat it or if it false, then the
       rabbit don't need to eat

       if carrot? [ask patches [set countdown random carrot-regrowth-time ;; initialize
       carrot grow clocks randomly set pcolor one-of [green brown]]]
```

```
Rabbit codes:

set-default-shape rabbit "rabbit"

       create-rabbit initial-number-rabbit  ;; create the rabbit, then initialize their
       variables [set color white

set size 1.5  ;; easier to see rabbit when enlarge to this size

set label-color blue - 2

set energy random (2 * rabbit-gain-from-food)

setxy random-xcor random-ycor]
```

Coyote codes:

```
set-default-shape coyotes "coyote"

        create-coyotes  initial-number-coyotes   ;;  create  the  coyotes,  then  initialize
        their variables

[set color black

set size 1.5  ;; easier to see

set energy random (2 * coyote-gain-from-food)

setxy random-xcor random-ycor]
```

Display carrot, rabbit and coyotes, and update plot to reflect overall picture:

```
display-labels

update-plot

end
```

Stop if there are no more carrot, rabbit and coyotes to create:

```
to go

if not any? turtles [ stop ]
```

Rabbit movement during simulation to eat carrot or die if no more energy:

```
        ask rabbit [move

        if carrot? [set energy energy - 1  ;; deduct energy for rabbit only if carrot?
        switch is on eat-carrot]

reproduce-rabbit

death]
```

Coyote movement during simulation to eat rabbit or die if out of energy:

```
ask coyotes [move

set energy energy - 1  ;; coyotes lose energy as they move

catch-rabbit

reproduce-coyotes
death]
```

Carrot growth and update environment to reflect color (green):

```
if carrot? [ask patches [ grow-carrot ] ]

tick

update-plot

display-labels

end
```

Rabbit and coyote movement random 50 left, right or fd:

```
to move  ;; turtle procedure

rt random 50

lt random 50

fd 1

end
```

Rabbit eat carrot procedure:

```
to eat-carrot  ;; rabbit procedure
;; rabbit eat carrot, turn the patch brown
if pcolor = green [
set pcolor brown
set energy energy + rabbit-gain-from-food  ;; rabbit gain energy by eating]
end
```

Rabbit reproduction procedure:

```
to reproduce-rabbit  ;; rabbit procedure
       if random-float 100 < rabbit-reproduce [   ;; throw "dice" to see if you will
       reproduce
       set energy (energy / 2)                     ;; divide energy between parent and
       offspring
        hatch 1 [ rt random-float 360 fd 1 ]   ;; hatch an offspring and move it forward
       1 step]
end
```

Coyote reproduction procedure:

```
to reproduce-coyotes  ;; coyote procedure
       if random-float 100 < coyote-reproduce [   ;; throw "dice" to see if you will
       reproduce
```

```
        set  energy  (energy  /  2)                     ;;  divide  energy  between  parent  and
        offspring
        hatch 1 [ rt random-float 360 fd 1 ]  ;; hatch an offspring and move it forward 1
        step]
end
```

Coyote catch rabbit procedure:

```
to catch-rabbit  ;; coyote procedure

let prey one-of rabbit-here                    ;; grab a random rabbit

if prey != nobody                             ;; did we get one?  if so,

[ ask prey [ die ]                          ;; kill it

set energy energy + coyote-gain-from-food ] ;; get energy from eating

end
```

Death procedure for rabbit and coyote:

```
to death  ;; turtle procedure

;; when energy dips below zero, die

if energy < 0 [ die ]

end
```

Carrot growth procedure:

```
to grow-carrot  ;; patch procedure

;; countdown on brown patches: if reach 0, grow some carrot

if pcolor = brown [

if else countdown <= 0

[ set pcolor green

set countdown carrot-regrowth-time ]

[set countdown countdown - 1 ]]

end
```

Update plot after each cycle of steps:

```
to update-plot

set-current-plot "populations"
```

```
set-current-plot-pen "rabbit"

plot count rabbit

set-current-plot-pen "coyotes"

plot count coyotes

if carrot? [set-current-plot-pen "carrot / 10"

       plot count patches with [pcolor = green] / 10  ;; divide by ten to keep it within
       ;; similar range as coyote and rabbit populations]

end
```

Display energy for carrot, rabbit and coyote if “show-energy” is on:

```
to display-labels

ask turtles [ set label "" ]

if show-energy? [

ask coyotes [ set label round energy ]

if carrot? [ ask rabbit [ set label round energy ] ]]

end
```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] Carlos Garcia-Montoro, Emilio Vivancos, Ana Garcia-Fornes, and Vicent J. Botti, (2007). “A Software Architecture-Based Taxonomy of Agent-Oriented Programming Languages, Languages,” Methodologies and Development Tools for Multi-Agent Systems.

[2] Steven F. Railsback, Steven L. Lytinen and Stephen K. Jackson (2006). Agent-based Simulation Platforms: Review and Development Recommendations. Simulation Vol 82.

[3] Andrei Borshchev and Alexei Filippov (2004). From System Dynamics and Discrete Event to Practical Agent Based Modeling: Reasons, Techniques, Tools. The 22nd International Conference of the System Dynamics Society, Oxford, England.

[4] Nicholas R. Jennings, Katie Sycara and Michael Wooldridge (1998). A Roadmap of Agent Research and Development. Journal of Autonomous Agents and multi-Agent Systems, 1(1).

[5] Mark E. Nissen and Kishore Sengupta (2006). “Incorporating Software Agents into Supply Chains: Expermental Investigation with a Procurement Task.” MIS Quarterly Vol 30 No. 1.

[6] Nelson Minar, Roger Burkhart, Chris Langton and Manor Askenazi (1996). “The Swarm Simulation System: A Toolkit for Building Multi-Agent Simulations.” Available at http://www.santafe.edu/research/publications/wpabstract/199606042 (Mar 2008).

[7] C.J.E Castle and A.T. Crooks (2006). Principles and Concepts of Agent-Based Modelling for Developing Geospatial Simulations. UCL Working Papers Series (110-Sep).

[8] W.N. Reynolds and D.S. Dixon (2000). paper presented at Complex Systems and Policy Analysis: New Tools for a New Millennium, RAND Science & Technology Policy Institute, Arlington VA, Sept 27-28, 2000, http://www.leastsquares.com/papers/rand2000.pdf (Mar 2008).

[9] W.N. Reynolds and D.S. Dixon (2003). The BASP agent-based modeling framework: applications, scenarios and lessons learned. System Sciences, 2003. Proceedings of the 36th Annual Hawaii International Conference on Volume 3, 6-9 Jan. 2003

[10] W.N. Reynolds and D.S. Dixon (2001). The Archimedes Combat Modeling Platform. Reprint from *Maneuver Warfare Science 2001*. G. Horne, M. Leonardi, Eds. (USMC, Quantico, VA, 2001).

[11] Douglas Druckenmiller, William Acar and Marvin Troutt (2004). Agent-Based Modeling and Simulation of Strategic Scenarios with REPAST 2.0. Paper submitted to Swarmfest 2004.

[12] Sean Luke, Claudio Cioffi-Revilla, Liviu Panait and Keith Sullivan (2004). Available at http://www.cs.gmu.edu/~eclab/projects/mason/publications/SwarmFest04.pdf.

[13] A.T. Crooks (2006). The Repast Simulation/Modelling System for Geospatial Simulation. UCL Working Papers Series (123-Sept).

[14] Hermant K. Bhargava, Ramayya Krishnan, Stephen Roehrig, Michael Casey, David Kaplan and Rudolf Muller (1997). Model Management in Electronic Markets for Decision Technologies: A Software Agent Approach. HICSS: Proceedings of the 30th Hawaii International Conference on System Sciences: Advanced Technology Track - Volume 5.

[15] AnyLogic website at http://www.coensys.com/anylogic.htm?gclid=CMXn7qWNoY8CFQLSYAodv1c4Zg (Mar 2008).

[16] Eric Bonabeau (2002). Agent-based modeling: Methods and techniques for simulating human systems. PNAS Vol 99, Suppl 3. Available at http://www.pnas.org/cgi/content/full/99/suppl_3/7280 (Mar 2008).

[17] SwarmWiki at http://www.swarm.org/wiki/Main_Page (Mar 2008).

[18] MASON website at http://cs.gmu.edu/~eclab/projects/mason/ (Mar 2008).

[19] Agent based model. Wikipedia at http://en.wikipedia.org/wiki/Multi-agent_systems (Mar 2008).

[20] Repast website at http://repast.sourceforge.net/index.html (Mar 2008).

[21] Leigh Tesfatsion (2008). General Software and Toolkits. Available at http://www.econ.iastate.edu/tesfatsi/acecode.htm (Mar 2008).

[22] Agent based modeling. Scholarpedia at http://www.scholarpedia.org/article/Agent_based_modeling (Mar 2008).

[23] Miles T. Parker (2008). What is Ascape and Why Should You Care? Available at http://ideas.repec.org/a/jas/jasssj/2000-13-1.html (Mar 2008).

[24] Mario E. Inchiosa and Miles T. Parker (2002). Overcoming design and development challenges in agent-based modeling using ASCAPE. PNAS Vol 99, Suppl 3. Available at http://www.pnas.org/cgi/content/full/99/suppl_3/7304 (Mar 2008).

[25] Coensys, Inc. System Dynamics Models Using AnyLogic 6. Available at http://www.coensys.com/system_dynamics.htm (Mar 2008).

[26] NetLogo website at http://ccl.northwestern.edu/netlogo/ (Mar 2008).

[27] Moshe Kress (2004). “A Social Structure Model for Evaluating the Effect Response Measures on the Spread of Smallpox.” Naval Postgraduate School, NPS-OR-05-002, Monterey, CA.

[28] Brian Hargrave (2008). “Integrated Data-Driven Decision Support Systems in a Laboratory Environment.” Naval Postgraduate School, TBD, Monterey, CA.

[29] R. Krishnan and K. Chari. Model management: survey, future research directions and a bibliography, Interactive Transactions of OR/MS 3 (1) (2000).

[30] Jade website at http://jade.tilab.com/ (Mar 2008).

INITIAL DISTRIBUTION LIST

1. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

2. Professor Dan Dolk
   Department of Information Sciences
   Naval Postgraduate School
   Monterey, California

3. Albert “Buddy” Barreto
   Department of Information Sciences
   Naval Postgraduate School
   Monterey, California

4. Christian Nguyen
   Port Arthur, Texas